

Harpy Tutorial

Martin Grabmüller Dirk Kleeblatt

April 27, 2007

Abstract

We present some of the features of Harpy, an in-line assembler for Haskell. This little tutorial shows how to write assembler programs without making one's hands dirty, staying in the beautiful pure functional world of Haskell. During this tutorial, we develop step by step an assembler implementation of the factorial function, and show how assembler code can be called from ordinary Haskell code.

This document is written as a literate Haskell program, so you can compile and run it without any modifications.

1 Introduction

To make use of Harpy and (a subset of) the x86 assembler instructions, we need to import some modules.

```
0 | import Harpy.CodeGenMonad
1 | import Harpy.X86Assembler
2 | import Foreign
3 | import Control.Monad.Trans
```

The module `Harpy.CodeGenMonad` defines the polymorphic type `CodeGen e s`, which is an instance of the `Monad` class. The type parameters `e` and `s` can be instantiated to the type of an user's environment and state, respectively. These behave like the environments and states known from the `Reader` and `State` monads. Besides this monadic type, this module defines some functions to make use of code labels, and provides an interface to a disassembler.

The module `Harpy.X86Assembler` provides a subset of the x86 assembler instructions, e. g. `mov` for moving memory words around. These instructions are implemented as class methods, to allow different addressing modes without hassle.

We additionally import the module `Foreign`, since we need some low level types to exchange parameters and results with our assembler code, and `Control.Monad.MonadTrans` to have some instances available.

2 A fast factorial function

Now we are ready to define the factorial function in assembler code.

```
4 | fac :: CodeGen e s ()
5 | fac = do loopTest ← newLabel
6 |           loopStart ← newLabel
```

```

7 |         ensureBufferSize 160
8 |         push ecx
9 |         mov ecx (Disp 8, esp)
10 |        mov eax (1 :: Word32)
11 |        jmp loopTest
12 |        loopStart @@ mul ecx
13 |        sub ecx (1 :: Word32)
14 |        loopTest @@ cmp ecx (0 :: Word32)
15 |        jne loopStart
16 |        pop ecx
17 |        ret

```

We first create two labels, `loopTest` and `loopStart`, to mark the test and the beginning of a loop. In lines 5 and 6, these labels are merely announced to Harpy, they are not (yet) defined to sit at a specific code position.

Line 8 saves the `ecx` register on the system stack, because we will use it as a loop counter, and want to restore it before returning to Haskell functions.

Line 9 shows an indirect addressing with displacement. Note, that all Harpy functions use Intel assembler style, i. e. the first operand is the destination and the second one the source of each instruction. So this line moves the memory contents at address `esp+8` into `ecx`. Since we will make a C call into our assembler code, this accesses the first parameter on the stack. When returning to the Haskell world via `ret`, we leave our result in `eax`, again adhering to the C calling convention.

The rest of `fac` just accumulates the factorial in `eax` while counting down `ecx`. Lines 12 and 14 show how our labels `loopStart` and `loopTest` are placed at specific code positions.

The function `fac` is not really our wanted factorial function. Instead it is a monadic command that, when executed, writes assembler code into a buffer. To ensure, that this buffer is always large enough to hold the generated instruction, you have to sprinkle your code with calls to `ensureBufferSize`. In line 7 we make sure that 160 bytes are available, which is enough for our 10 instructions. As a rule of thumb, no instruction can be larger than 16 bytes, so the number of assembler instructions times 16 is a safe upper bound.

The next section shows how to prepare a call into such a buffer.

3 Preparing a call

The module `Harpy.Call` defines some functions to call functions written in assembler with various argument and result types. But since it is possible to use all types suitable for FFI calls as argument or result, sooner or later you will need some combination not yet implemented. So here we show how to define your own calling stub.

```

18 | $(callDecl "callFac" [t|Word32 → Word32|])

```

The Template Haskell function `callDecl` is used to declare a function `callFac` which will call our assembler fragment. We want to pass a parameter of type `Word32`, and expect a result of the same type, that's why we give `Word32 → Word32` as argument to `callDecl`. If you wonder about the fancy `$` and `[t|]`, either look them up in the Template Haskell documentation, or just ignore them. However, to make this line compile,

you have to switch on Template Haskell, which is done for the Glasgow Haskell compiler by the command line flag `-fth`.

4 Calling fac

Now we have all we need to call into our factorial function.

```
19 runFac :: CodeGen e s ()
20 runFac = do fac
21           x ← liftIO readLn
22           y ← callFac x
23           liftIO (putStrLn (show y))
```

We first call `fac` to write our assembler code into the internal buffer. Since the `CodeGen` monad is an instance of `MonadIO`, we can use `liftIO` to use all commands we wish from the `IO` monad. Here, we use this to read the argument to the factorial function from the keyboard, and to write the result back to the screen. Line 21 calls into the internal code buffer with our assembler instructions using the stub declared in the last section.

5 How to use it

Up to now, all our functions live in the `CodeGen` monad. To make use of them, we have to *unlift* them into the `IO` monad. This is done by `runCodeGen`.

```
24 main :: IO ()
25 main = do
26   (finalState, result) ← runCodeGen runFac () ()
27   case result of
28     Right () → return ()
29     Left err → putStrLn (show err)
```

The second and third arguments to `runCodeGen` are the initial environment and state. Since we did not use them, their type is polymorphic and we can use `()` as initial values. The result is a pair consisting of the final state, and a result value. A value constructed with the constructor `Right` indicates a successful run, while `Left` values indicate runtime errors. These might occur for instance because of infeasible addressing modes.